LA-UR-21-26058
Approved for public release; distribution is unlimited.

Title: Pagosa Performance on a HPE Apollo80 System

Author(s): Graziano, Vincent John
Nystrom, William David
Pritchard, Howard Porter Jr.
Smith, Brandon Michael
Gravelle, Brian Joseph

Intended for: random presentation

Issued: 2021-06-28

Pagosa Performance on a HPE Apollo80 System

Vince Graziano
David Nystrom
Howard Pritchard
Brandon Smith
Brian Gravelle

7/13/21

LA-UR-YY-XXXXX

Presentation Outline

- Description of Pagosa and Shaped Charge
- Running Pagosa on HPE Apollo 80 A64FX node
- Performance limitations
- Role of Compilers
- Code refactoring to improve performance
- Compare to other Processor types
- Conclusions

Pagosa

- 3 dimensional, multi-material shock wave physics code
- Uses a structured cartesian mesh
- Explicit finite difference method in the Eulerian frame used to solve equations of motion, etc.
- Material equations-of-state (EOS) can be evaluated analytically or via tabular lookup
- Written in Fortran (F2003), makes extensive use of array syntax and Fortran intrinsics
- Parallelism – MPI only
- Uses OpenMP for GPU offload (not subject of this talk)

Running Pagosa on Apollo 80 A64FX node

- One socket of A64FX with 48-cores with 4 numa nodes (CMGs)
- ARMv8.2-A+SVE SIMD width of 512-bits
- 32GBs of HBM2 memory (8 GB/CMG), no L3 cache, one L2 cache per CMG

- Shaped Charge problem with 25-materials
  - 1 mm resolution, 3D cartesian mesh
  - Mixture of analytic and tabular equations-of-state
  - Typical of actual user problems
- Compilers:
  - CCE 10.0.2
  - ARM 20.2.1
  - GNU 10.2.0
  - Fujitsu 4.5.0 (only recently obtained)

Performance Limitations

- Coded largely in Fortran array-syntax
  - Difficult for compilers to optimize well
  - Each array-syntax statement implies operations and bandwidth
- Depending on mesh size, data is streaming to and from LLC or memory
  - In the case of A64FX, data will stream from HBM2
- A64FX stats for CCE built version (from CrayPat):
  - 70.7% of instructions had backend stalls
  - 24.1% of instructions were SIMD
  - IRC of 0.56 (seen this for a variety of codes on A64FX)

Fortran Array-Syntax Patterns in Pagosa

```
real, dimension(0:mx,0:my,0:mz):: a,b,c,d,e
a = b * c
d = a + e
```

Semantically equivalent to

```
do k = 0, mz
  do j = 0, my
     do i = 0, mx
         a(i,j,k) = b(i,j,k) * c(i,j,k)
     enddo
  enddo
enddo
do k = 0, mz
  do j = 0, my
     do i = 0, mx  ! reuse “a” from where?
         d(i,j,k) = a(i,j,k) + e(i,j,k)
     enddo
  enddo
enddo
```

Fortran Array-Syntax Patterns in Pagosa

- What should a compiler do to improve performance?

- If compiler fuses all 3-loops, “a” can be reused from a vector register instead of memory or cache

```
do k = 0, mz
  do j = 0, my
     do i = 0, mx
         a(i,j,k) = b(i,j,k) * c(i,j,k)
         d(i,j,k) = a(i,j,k) + e(i,j,k)
     enddo
  enddo
enddo
```

Fortran Array-Syntax Patterns in Pagosa

If the compiler can collapse the loops into a single loop-nest:

- reduces loop-overhead
- improves vector efficiency, esp with strong scaling
- CCE does extensive loop-collapse in Pagosa, Fujitsu less so

```
do i = 0, (mx+1)*(my+1)*(mz+1)
    a(i,0,0) = b(i,0,0) * c(i,0,0)
    d(i,0,0) = a(i,0,0) + e(i,0,0)
enddo
```

Role of Compilers on A64FX

- Why CCE and Fujitsu compilers generate so much better code than ARM compiler (1.9x faster for CCE, 1.7 times faster for Fujitsu)
- Both compilers are better at vectorization overall than ARM LLVM, and
  - CCE:
    - Significant fusion of array-syntax statements
    - More Vectorization of loops/array-syntax
    - Loop-collapse
    - 512-bit fixed style of vector-code
  - Fujitsu:
    - Avoid branch prediction using predication for SVE ops
    - Software pipeliner (not sure about this yet)
    - Loop Fissioning (not sure about this yet)
- ARM compiler does:
  - Limited fusion of array-syntax statements
  - Much less vectorization
  - no loop-collapse
  - Vector-length-agnostic (VLA) vector-code

Example of CCE optimization for Array-Syntax

Key: V – vectorized, f – loop-fusion, C – loop-collapse

```
52.  fVC----<>   Tmp1(:,:,:) = (Grad(:,:,:,1,1) + Grad(:,:,:,2,2) + Grad(:,:,:,3,3))
53.
54.  f------<>   dA(:,:,:) = (Grad(:,:,:,1,1) - Tmp1(:,:,:)) * dt
55.  f------<>   dB(:,:,:) = (Grad(:,:,:,2,2) - Tmp1(:,:,:)) * dt
56.  f------<>   dC(:,:,:) = (Grad(:,:,:,3,3) - Tmp1(:,:,:)) * dt
57.  f------<>   dD(:,:,:) = (.5 * (Grad(:,:,:,1,2) + Grad(:,:,:,2,1))) * dt
58.  f------<>   dE(:,:,:) = (.5 * (Grad(:,:,:,1,3) + Grad(:,:,:,3,1))) * dt
59.  f------<>   dF(:,:,:) = (.5 * (Grad(:,:,:,2,3) + Grad(:,:,:,3,2))) * dt
60.
61.  f------<>   W1(:,:,:) = (Grad(:,:,:,1,2) - Grad(:,:,:,2,1)) * dt2
62.  f------<>   W2(:,:,:) = (Grad(:,:,:,1,3) - Grad(:,:,:,3,1)) * dt2
63.  f------<>   W3(:,:,:) = (Grad(:,:,:,2,3) - Grad(:,:,:,3,2)) * dt2
```

Compare A64FX to other node types

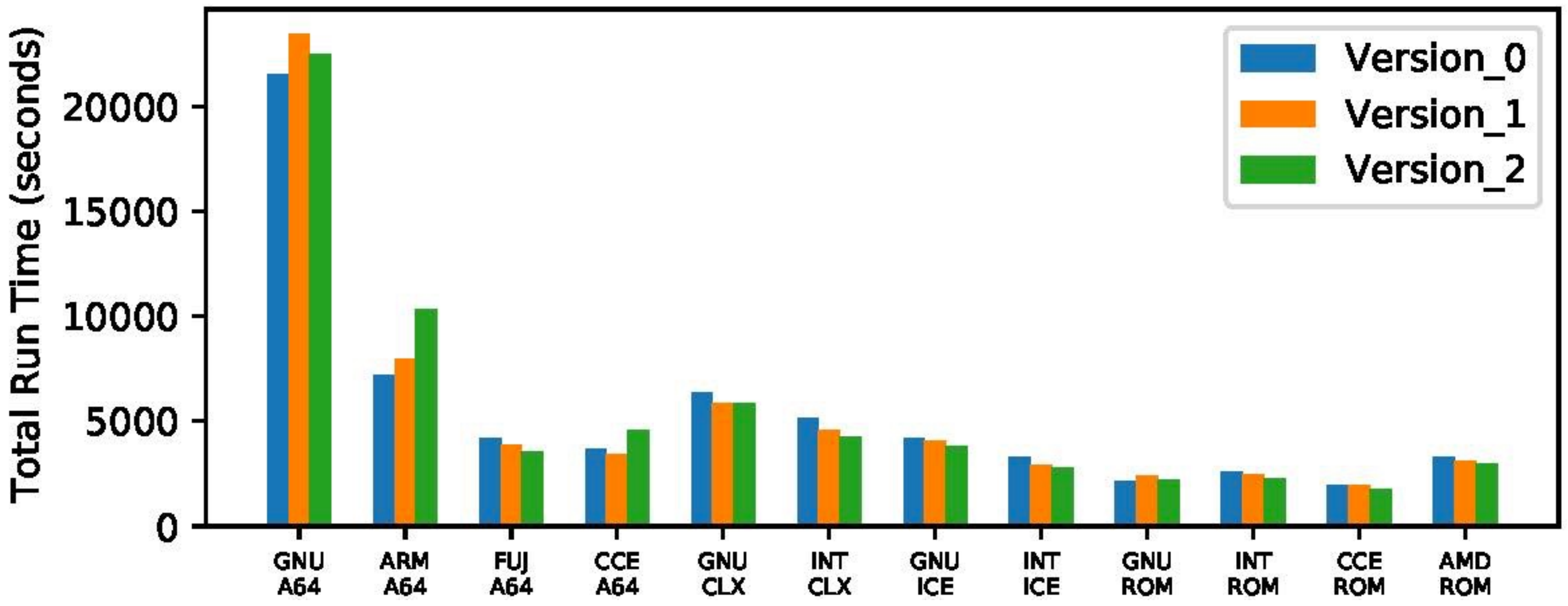
- AMD Rome with 2-sockets/node and 128-cores of AVX2
  - Using 4 compilers: CCE, Intel, AOCC, and GNU
- Intel Xeon Cascade Lake with 2-sockets/node and 48-cores of AVX512
  - Using Intel and GNU compilers
- Intel Xeon Ice Lake with 2-sockets/node and 48-cores of AVX512
  - Using Intel and GNU compilers

Could source changes help performance?

- Version 1:
  - Take selected array-syntax statements and recode as loops
    - Kernels from routines high in profile
    - Manual loop-fusion of recoded loops to get data reuse
    - To make up for compiler optimization NOT doing it
- Version 2:
  - Manually inline most expensive routines into the calling routine
  - Manually fuse loops from these routines into single, more compute intensive loops
  - Replace vector temporaries with scalars

- Answer: Yes, such source changes can help for some compilers

Timing Results



A64 – A64FX
CLX – Cascade Lake
ICE – Ice Lake
ROM – AMD Rome

Conclusions

- Pagosa performance is dependent on the compiler ability to:
  - Vectorize array-syntax well
  - Loop-fusion of array-syntax statements
  - Loop-collapse
  - CPU Vendor compiler specific optimizations
- A node of Apollo 80 with A64FX socket performed:
  - Almost 2x faster built with CCE or Fujitsu compared to ARM compiler
  - Better than a node of Xeon Cascade Lake
  - Slightly worse than a node of Xeon Ice Lake
  - Worse than a node of AMD Rome probably because of core-count disadvantage and Rome’s large L3 cache
- Making selected source changes can help compilers

Next Steps

- More detailed investigations of Fujitsu compiler capabilities:
  - Investigate use of loop fission for compute intensive loops
  - Investigate performance impact of software pipeliner
- Investigate performance of an ALE application on A64FX